Design, Implementation, and Evaluation of Task Management in Distributed Fault-Tolerant Real-Time Systems

Project Investigator: *Chao-Ju (Jennifer) Hou*

Graduate Students: *Bin Wang, Hung-ying Tyan, and Yi Ye*

Dept. of Electrical Engineering

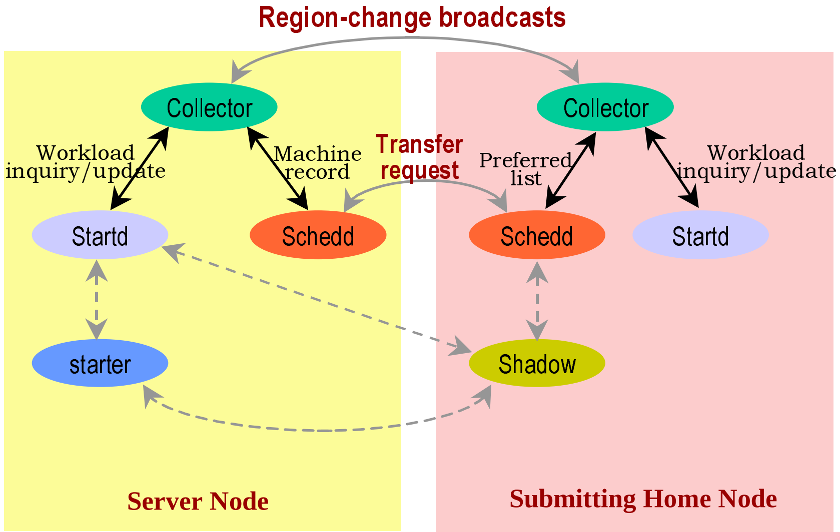
The Ohio State University

Columbus, OH 43210-1272

jhou@ee.eng.ohio-state.edu

http://eewww.eng.ohio-state.edu/drcl

DAEMONS IN THE DECENTRALIZED LS MECHANISM



NEW IDEAS

•Design, implementation, and empirical evaluation of a task management system in distributed real-time environments to meet the timeliness requirements of both periodic and aperiodic tasks.

•Incorporation of fault tolerance by (i) identifying and replicating critical module; (ii) deploying checkpointing and rollback recovery technology; and (iii) coordinating workstations to restart checkpointed processes in case of failure.

•Implementation of the proposed mechanism as a *software* layer that lies between OS and application processes.

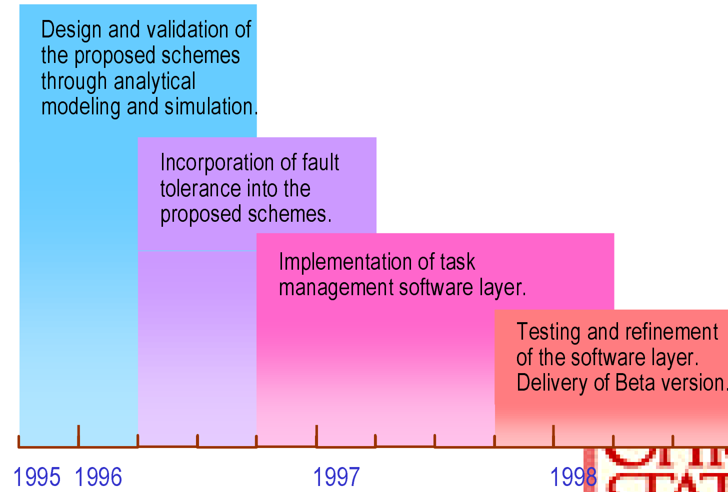
IMPACTS

•The project is a combination of two synergistic components: scheme development in a well-defined analytic framework and validation with software system building and experiments.

•Resulting software can be readily ported to POSIX-compliant operating systems and provide user-transparent task management services.

•Many problems investigated, e.g., how to insulate the internals of OS from application programs to achieve interoperability, how to provide distributed services in the form of system calls, are consistent with the objectives pursued by military/industry sectors.

SCHEDULE



Outline of Presentation

- *Project Overview*
  - Real-time task management
  - Objectives
- *Fault-tolerance Components in Our Project*
  - Replication of critical task modules
  - Load redistribution
  - Checkpointing and rollback recovery
- *Software Implementation and Status*

Real-Time Task System

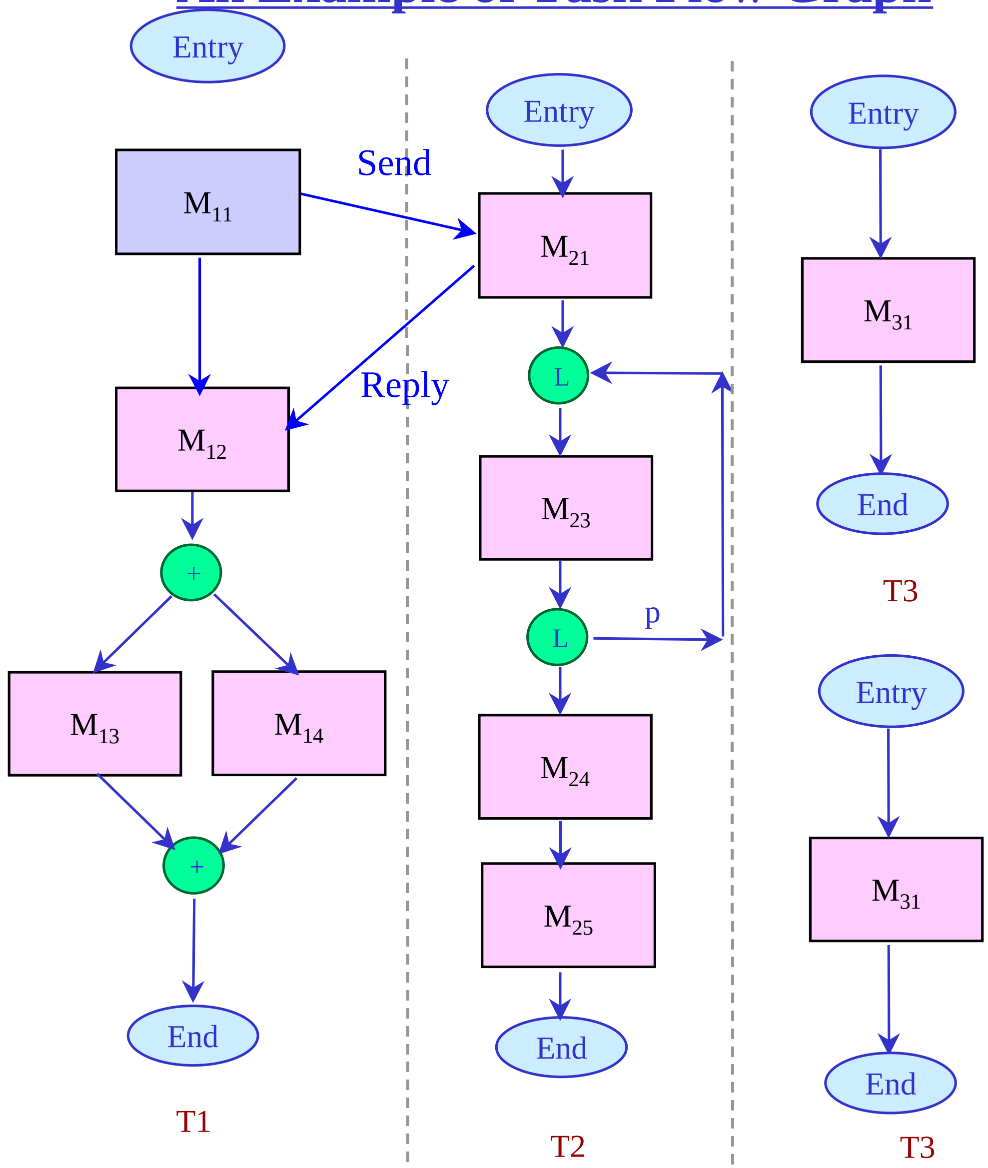
Every task is characterized by a laxity -- the latest time a task must start execution in order to meet its deadline.

- *Periodic tasks*
  - Invoked at fixed time intervals.
  - Attributes are usually known *a priori*.
- *Aperiodic tasks*
  - Invoked randomly in response to environmental stimuli.
  - Attributes are not completely specified.

Methodology Used

- *Task decomposition*: Decompose periodic tasks into a set of communicating modules, and represent them by a task flow graph.
- *Module allocation*: Allocate periodic task modules to workstations subject to precedence constraints and timing requirements.
- *Load redistribution*: Dynamically redistribute aperiodic tasks as they arrive to minimize the probability of dynamic failure.
- *Scheduling*: Schedule modules/tasks on a workstation using the rate-monotonic policy, the earliest-deadline-first policy, or variations thereof.

An Example of Task Flow Graph



Replication of Critical Task Modules

• We devise a *module allocation* scheme to allocate periodic task modules in a *planning cycle* so that

– each task can be completed with *both* logical and timing correctness.

– task precedence and timing constraints are satisfied.

• With the *critical path analysis* approach, we devise a *module replication* scheme to identify and replicate modules whose completion is critical to the timely completion of tasks in the system.

Results were reported in

(1) C. Hou and K. Shin, "Allocation of periodic task modules with precedence and deadline constraints in distributed real-time systems," *IEEE Trans. on Computers*, Vol. 46, No. 12, 1997 (19 pages).

(2) C. Hou and K. Shin, "Replication and allocation of task modules in distributed real-time systems," *IEEE 25th Annual Int'l Symp. on Fault-Tolerant computing*, pp. 26--35, June, 1995.

Module Replication for Fault Tolerance

Given a task flow graph that describes the computation and communication modules and the precedence and timing constraints among them, we consider

- which modules are replicated;
- how many copies are replicated for each selected module;
- how to assign the replicas to workstations;
- how to schedule the replicas on each workstation.

with the objective of achieving timely correctness.

Critical Path Analysis

- Observation: There is no need to replicate modules that can be completed in time even with consideration of worst-case recovery time.
- Criterion for selecting critical modules:

$$LC_i - r_i < e_i + t_r ,$$

then M_i may not be completed in time in the case of failure.

where

- LC_i is the latest completion time of module $M_{i,}$
- r_i is the earliest release time of $M_{i,}$
- e_i is the execution time of $M_{i,}$
- t_r is the worst-case error recovery time.

Critical Path Analysis

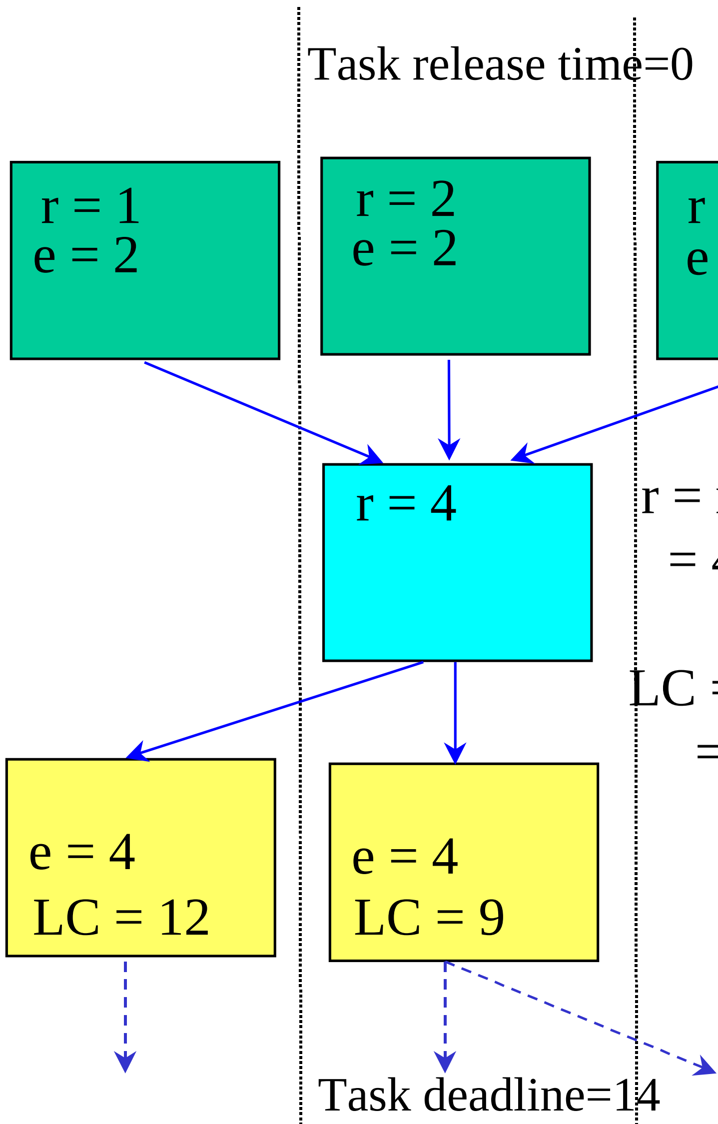
- Key Step 1: Calculate r_i from (1) the invocation time of the task and (2) the precedence constraints preceding M_i.
- Set r_i initially to the invocation time of the task to which M_i belongs. Then, modify r_i as

$$r_i = \max \{ r_i , \max_j \{ r_j + e_j : M_j \dashrightarrow M_i \} \}$$

- Key Step 2: Calculate LCi from (1) the deadline of the task and the precedence constraints after Mi.
- Initially set LC_i to the deadline of the task to which M_i belongs. Then, modify LC_i as

$$LC_i = \min \{ LC_i , \max_j \{ LC_j - e_j : M_i \dashrightarrow M_j \} \}$$

Example of Critical Path Analysis



r = max {0, max{1+2, 2+2, 3+1}}
  = 4

LC = min{14, min{12-4, 9-4}}
  = 5

Determination of #Replicas

- There is a tradeoff between fault tolerance and timing requirements:
  - The larger #replicas, the better fault-tolerance capability.
  - Excessive replicas may jeopardize the timely completion of modules.
- We augment the task system with m replicas for each selected critical module, and use the module allocation scheme, coupled with the module scheduling algorithm, to determine the assignment and scheduling of all modules.
- If there is computation power left, try to increase #replicas until the required probability of dynamic failure is violated.

Load Redistribution

- We characterize load redistribution (or load sharing, LS) with three component policies: the *transfer policy*, the *location policy*, and the *information policy*, and devise policies so that

  (1) an overflow task will not be transferred to an "incapable workstation;"

  (2) multiple nodes will not send their overflow tasks to the same workstation;

  (3) communication overhead is kept minimal;

  (4) nodes are coordinated with one another to restart checkpoint files in case of failures.

Research results were reported in

(1) C. Hou and K. Shin, "Evaluation of load sharing with consideration of its communication activities," *IEEE Trans. on Parallel and Distributed Systems*, Vol. 7, No. 7, pp. 724--740, July, 1996.

(2) C. Hou and K. Shin, "Implementation of decentralized load sharing in networked workstations using the Condor package," *Journal of Parallel and Distributed Systems*, Vol. 40, pages 173--184, January, 1997.

Transfer Policy

- We characterize the workload (i.e., state) in terms of

  (1) average workload in the past 5 minutes (**AvgLoad**);

  (2) keyboard idle time (**KeyboardIdle**);

  (3) the number of tasks queued and their states (**Task_state**).

- A workstation is evaluated to be

  Idle: (**AvgLoad** **<=** 0.3) && (**KeyboardIdle** >= 15 min) &&
  (**Task_state** = = NoTask)

  Other states, e.g., lightly-loaded, moderately-loaded, heavily-loaded can be similarly defined and configured.

- Upon arrival of a task, if the local workstation is not idle or lightly-loaded, then the task is considered for transfer.

Information Policy

- We use *state-change broadcasts* as the information policy to provide each workstation with timely state information.
- A workstation broadcasts a message, informing all the other workstations of its state change, whenever its state switches from one to another.
- Message exchange occurs *only when the state changes*, and thus the communication overhead is reduced while the state information kept at each workstation is likely up-to-date.
- Fine-tuning is possible by, for example, having a workstation broadcast a message when its state changes from idle to medium-loaded and vice versa.

Location Policy

- We use *preferred lists* to coordinate workstations on their location policy: Each workstation orders all the other workstations into a preferred list subject to:
  - a workstation is the *k*th preferred workstation of one and only one workstation.
  - if workstation *i* is the *k*th preferred workstation of *j*, then workstation *j* is also the kth preferred workstation of *i*.

  We prove that *load balance is achieved in the long run with the preferred list*
- Whenever a workstation decides to transfer a task, it traces its preferred list, and locates the first idle/lightly-loaded workstation in the list.
- The preferred list of a workstation is static, but the states associated with the workstations in the list may change.

Fault-Tolerance in Location Policy

In the case of workstation failure

- If we simply drop the failed workstation from the preferred list of a workstation, the desired load balance feature is destroyed.

  Solution: we adjust the preferred lists in case of failure to retain the two properties of the preferred lists.

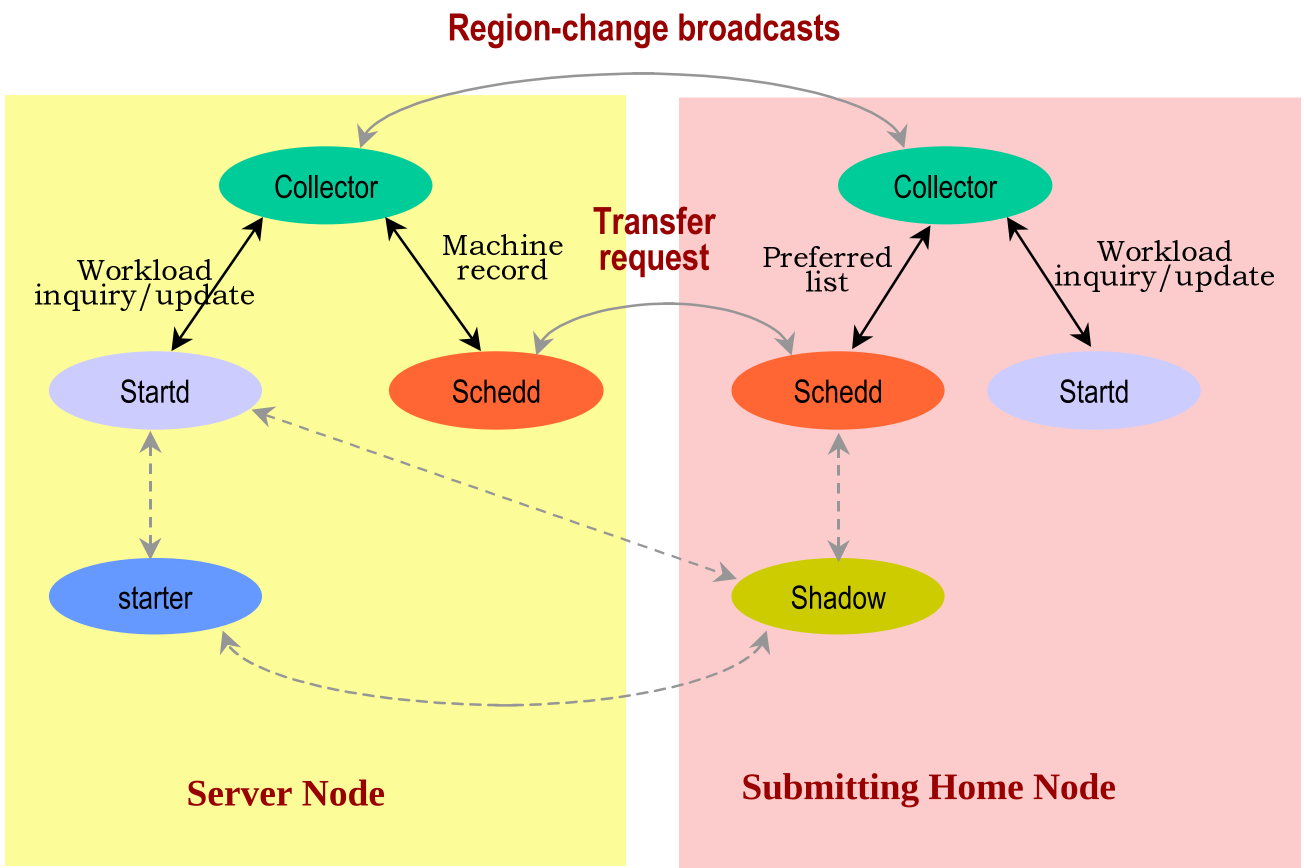
- If no checkpoint file is taken and stored somewhere else, all the tasks executed at a failed workstation will be lost.

  Solution: we devise an approach to coordinate workstations to keep backup checkpoint files for their most preferred nodes so that tasks executed/queued at a failed workstation can be restarted whenever needed.

Software Configuration

- We implement the load sharing mechanism as a software layer outside the OS kernel at the user level, since this design
  - eliminates the need to access/change the internals of OS,
  - allows us to concentrate on varying the degree of design complexity and
  - is portable and can be ported to any POSIX-compliant platforms.
- Based on the Condor software distributed by Univ. of Wisconsin -- Madison, we configure the proposed mechanism into three daemons, *Collector*, *Schedd*, *Startdd,* which constantly run on each workstation.
- Two additional processes, *Shadow* and *Starter*, run on the submitting workstation and the server workstation, respectively, when a task is remotely executed.

Daemon Configuration



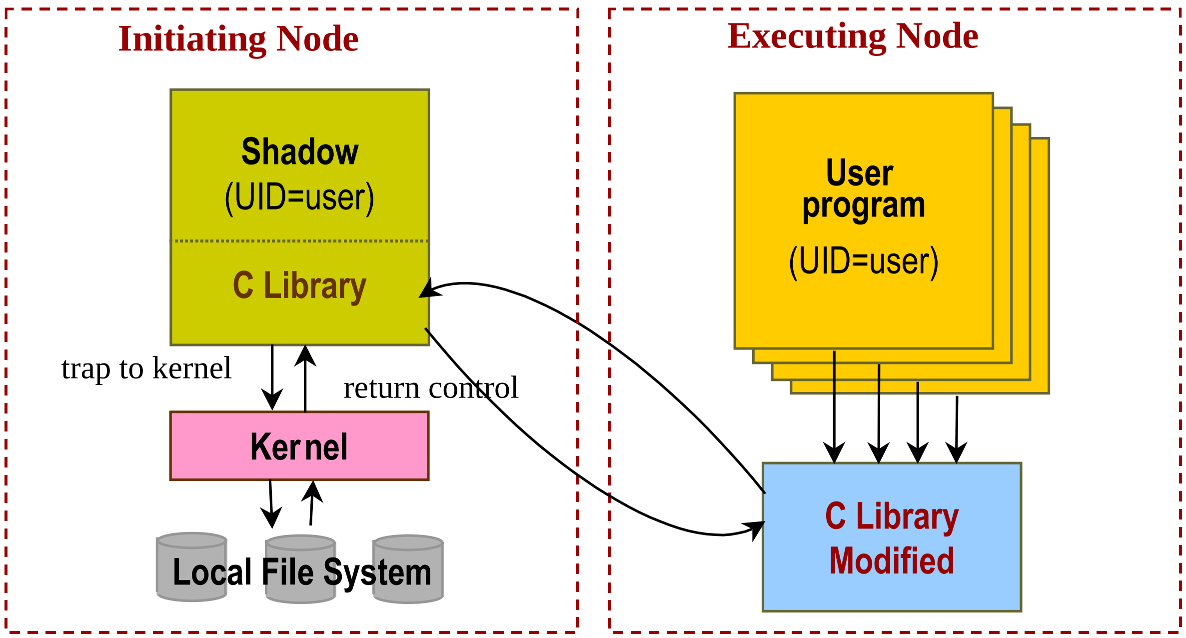
Implementation Features

- Both module allocation and load sharing are performed *transparently* to users.
- No code change is needed for user programs; only a relink to the modified C library is required for user programs.
- We preserve local execution environment for remotely executing processes via *remote system call mechanism*.
- We set protection for local file systems; they will not be touched by remotely executing tasks.
- *Remote processes are periodically checkpointed, sen back to the home workstation, and restarted at some other workstation in the case of failure.*
- A Java-integrated monitor is provided to facilitate monitoring of participating workstations and submitted tasks.

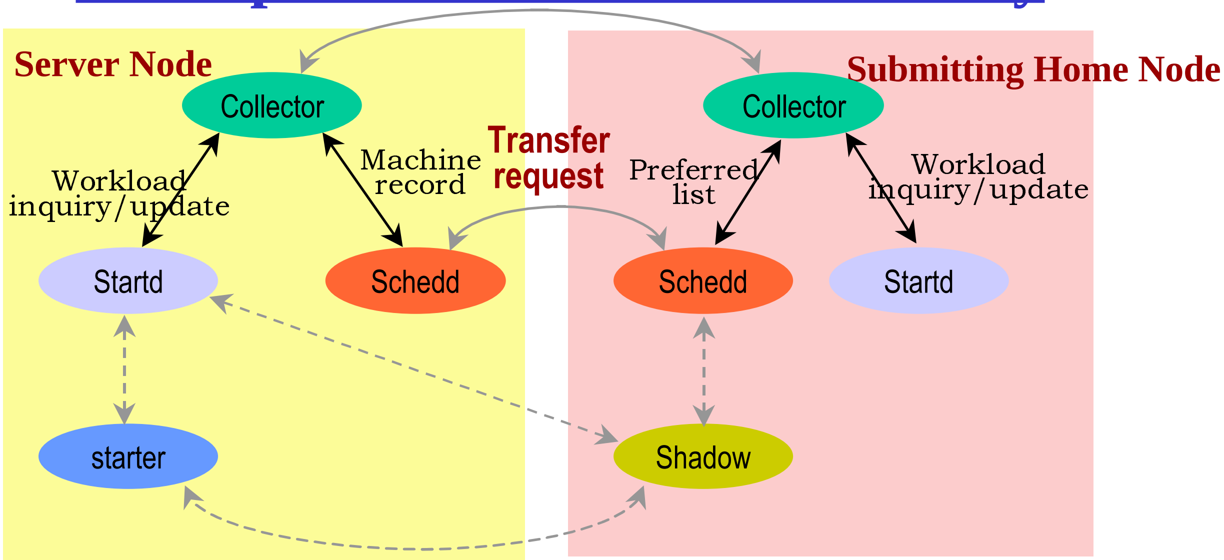
Remote System Calls

All environment-related system calls issued by a remote executing task are

- trapped by the *modified* C system call stubs, and
- forwarded to the *Shadow* on the home workstation which acts as an agent and executes the system calls on behalf of the task.
- The *Shadow* executes the system call on behalf of the remotely executing task, packages up the call results, and sends them back to the system call stubs on the executing machine.



Checkpoint and Rollback Recovery



- The state of a process is transferred in the form of checkpoint files. Before a process is executed for the first time, its executable file is augmented to a checkpoint file without stack area.
- Starter causes a running task to save its file state and stack and then dump core. Starter then creates a new checkpoint file from pieces of the previous checkpoint and a core image.
- Starter sends the checkpoint file to the Shadow at the home workstation.
- Shadow at the home workstation monitors whether or not the remote starter is alive by periodically probing; in the case of starter failure, Shadow will restart the checkpoint file by treating it as a newly-arrived task.

Details on Checkpoint File Creation

- *Creating checkpoint file:*

  (1) A special version of *crt()* is included which sets up *CKPT()* as the signal handler of *SIGTSTP*.

  (2) *Starter* causes a running task to checkpoint by sending it the signal *SIGTSTP*.

  (3) When *CKPT()* is called, it updates the table of open files by recording the current file positions. A *setjump* is executed to save key register contents in a global data area, and the process sends itself a *SIGQUIT* signal which results in a core dump.

  (4) The *Starter* then combines the executable file and the core file to produce a checkpoint file.

- *Restarting checkpoint file*:

  (1) A restarted checkpoint file starts from the special *crt()* which sets up *restart()* as the signal handler for *SIGUSR2* and sends the checkpoint process that signal.

  (2) When *restart()* is called, it reads the saved stack in from the checkpoint file, reopens and repositions all the files, and execute a longjmp back to *CKPT()*.

  (3) *CKPT()* returns to whatever code segment at the time of checkpoint.

Implementation Status

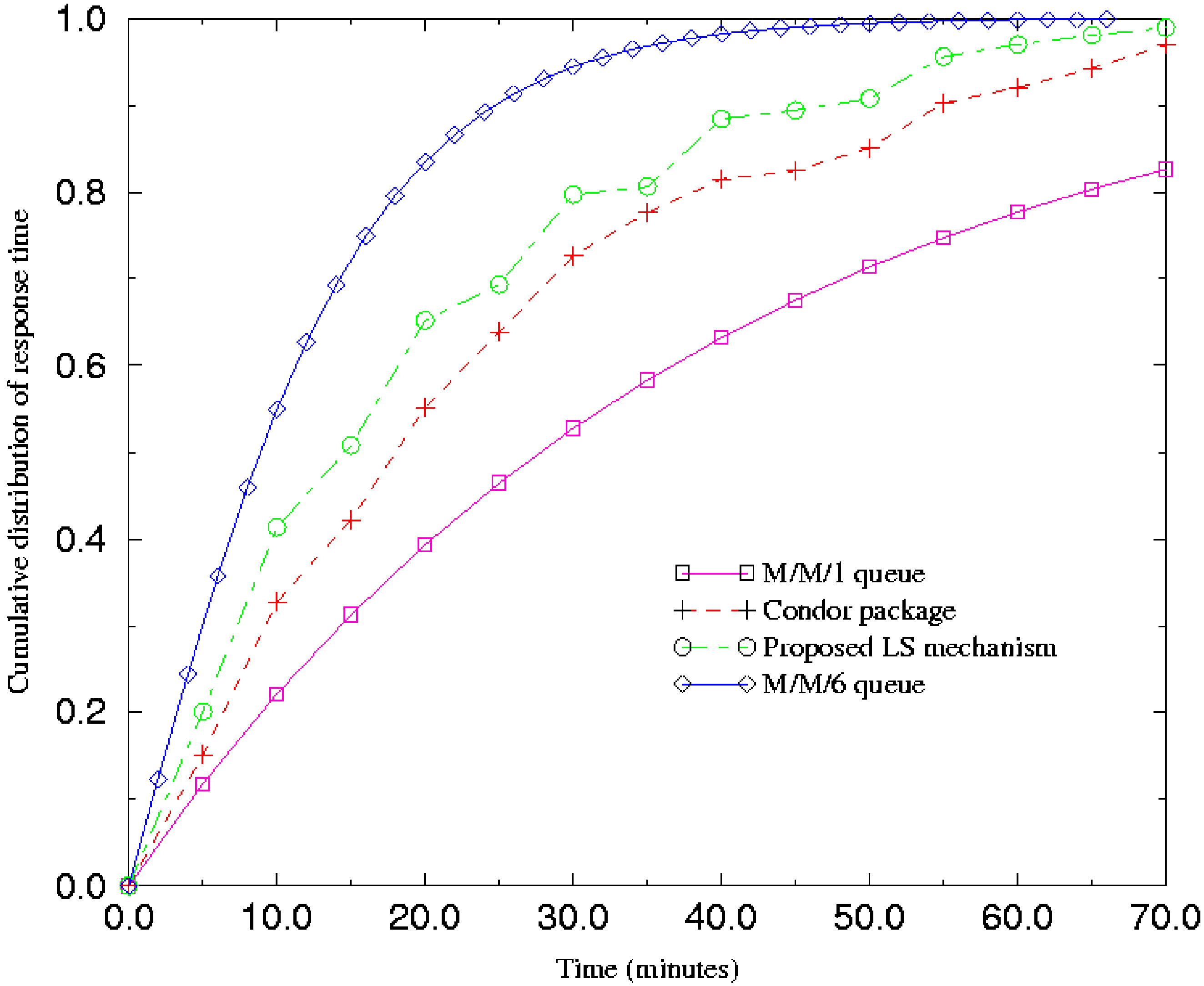
- The first version of the software prototype is up and running, and can be ported to most UNIX environments. But,

  (1) It does not support inter-process communication (IPC) and signal facilities as a result of placing the task management mechanism outside the OS kernel.

  (2) A consequence of (1) is that although we devise a checkpointing scheme that *dynamically* varies checkpoint interval with respect to IPC frequency to avoid state inconsistency (i.e., the received-but-not-yet-sent scenario) and to reduce process rollback propagation, it is virtually of no use now.

- For demonstration purpose, we are currently implementing a Java-integrated graphic user interface that will probe the Monitor daemon on each workstation for task execution statistics.

  The Java program can be a standalone application or a Java applet readily to be incorporated into a web page; and (2) can be remotely invoked (i.e., not necessarily on one of the workstations)


1.0
0.8
0.6
0.4
0.2
0.0
0.0
10.0
20.0
30.0
40.0
50.0
60.0
70.0
Cumulative distribution of response time
Time (minutes)
M/M/1 queue
Condor package
Proposed LS mechanism
M/M/6 queue

How Our Research Results Can be Used

- JPL has already ftp'ed our first software release (that contains the load redistribution and checkpoint/rollback recovery components) to their site, and plans to port the software to their testbed.
- We will provide technical support and student helps whenever needed.
- We will be glad to share our research results on design of module replication, checkpoint/rollback recovery schemes with JPL and/or other commercial/military sectors.
- We look forward to long-term collaboration with JPL.